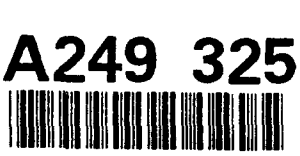

(2)

New Loop Transformation Techniques for Massive Parallelism

Lee-Chung Lu and Marina Chen

YALEU/DCS/TR-833
October 1990

Yale University
Department of Computer Science

New Loop Transformation Techniques for Massive Parallelism

Lee-Chung Lu and Marina Chen

YALEU/DCS/TR-833
October, 1990

This paper presents new loop transformation techniques that can extract more parallelism from a class of programs than existing techniques. A formal mathematical framework which unifies the existing loop transformation techniques is given. We classify *schedules* of a loop transformation into three classes: *uniform*, *subdomain-variant*, and *statement-variant*. New algorithms for generating these schedules are given. Viewing from the degree of parallelism to be gained by loop transformation, the schedules can also be classified as *single-sequential level*, *multiple-sequential level*, and *mixed* schedule. We describe iterative and recursive algorithms to obtain multiple-sequential level and mixed schedules respectively based on the algorithms for single-sequential level schedules.

This work has been supported in part by the Office of Naval Research under Contract N00014-89-J-1906, N00014-90-J-1987.

1 Introduction

One of the central issues in restructuring compiler is to discover parallelism automatically and generate correct parallel control structures that can take advantage of the large number of processors. The advent of massively parallel machines opens up opportunities for programs that have large-scale parallelism to gain tremendous performance over those that do not. This paper presents new loop transformation techniques that can extract more parallelism from a class of programs than existing techniques. The particular class of programs are those that consist of perfectly nested loops possibly with conditional statements where the guards as well as the array index expression are affine expressions of the loop indices.

Organization of the Paper To make this paper self-contained, we describe the notations and terminologies of the basic concepts relating to data dependence and loop transformation in Section 2. We then present a formal mathematical framework which unifies the existing loop transformation techniques, and sets the stage for discussing the more general classes of *loop transformers* in Section 3. A *loop transformer* is a function that relates a given loop nest with its transformed version, and consists of two parts: a *spatial morphism*, and a *temporal morphism*, called a *schedule*. Next, in Section 4, we classify schedules by the properties of *uniformity* and the degree of parallelism to be gained, and describe the functional forms of the schedules for each class. Existing loop transformation techniques are given as examples of these classes of schedules.

In Section 5, we review Quinton's algorithm for obtaining *single-sequential level uniform* schedules and present the problem formulations for two new classes of schedules, namely, *subdomain schedules* and *statement-variant schedules* and the algorithms to generate them. The generation of subdomain schedules requires non-linear programming, and an alternative heuristic algorithm using linear programming is given.

Section 6 describes an iterative algorithm to obtain *multiple-sequential level* schedules based on the algorithms for single-sequential level schedules. Section 7 presents a recursive algorithm to generate *mixed* schedules that result in imperfectly nested loops, again using the algorithms for single-sequential level schedules as the basic step.

Finally, we illustrate the usefulness of the new loop transformation techniques with an example program in Section 8. Versions of the transformed program using different schedules are implemented on a Connection Machine CM/2. The difference in performance, which is essentially due to the available parallelism determined by the schedule, can amount to two orders of magnitude.

Previous Work Numerous techniques such as statement reordering, loop vectorization, interchanging, permutation and skewing used in restructuring compilers [1, 2, 3, 4, 5, 6, 7, 8, 9, 20, 21, 33, 34, 37], have been proven effective in gaining parallelism for vector computers and small-scale shared memory parallel machines.

Much work in the area of mapping recurrence equations to systolic architectures [11, 12, 15, 17, 19, 24, 26, 25, 27, 28, 29, 30], in contrast, focuses on developing algorithms for loop

skewing.

Guerra [14] and Lin [22] discussed different transformation functions over different subdomains of the the iteration space of a loop nest very similar to the subdomain schedules presented here. But they have not described any algorithms for generating such schedules.

Delosme [12] and Rao [31] discussed different transformation functions over different subsets of the set of loop statements. To date, there has not been any automated procedure for doing such a transformation.

In all previous work on loop transformation, dependence vectors and dependence direction vectors are all that are needed. And for the type of loop nests of interest, there are constant numbers of such vectors. In order to generate subdomain and statement-variant schedules, we need actually to capture the *dependence index pair* where a dependence relation occurs. The problem is that there are many such pairs that need to be considered, and they can be infinitely many when the loop bounds are unknown at compile time. We need to rely on a technique called *polyhedra decomposition* [13, 30, 32] to manage the complexity of the algorithm.

Some recent work attempting to formalize loop transformations requires the transformation functions to be *unimodular* [7, 8, 33]. We will show that this requirement is not essential, and allow a much more general class of functions to be used as a loop transformer.

2 Definitions and Terminologies

Throughout this paper, programming examples are written in a Fortran-like notation although the transformation techniques also apply to functional languages.

Index Domains Let $[a, b]$ be an *interval domain* of integers from a to b. We define an *index domain* D (also called an *iteration space* in [34]) of a d-level *perfectly nested* loop

Loop Nest 1

$$
\begin{array}{l}
\mathsf{DO}\ (i_1 = l_1, u_1)\,\{ \\
\quad \mathsf{DO}\ (\ldots)\,\{ \\
\quad\quad \mathsf{DO}\ (i_d = l_d, u_d)\,\{ \\
\quad\quad\quad \mathit{body}\,\}\qquad\quad \}\,\}
\end{array}
$$

to be the Cartesian product $[l_1, u_1] \times \ldots \times [l_d, u_d]$ of d interval domains $[l_k, u_k]$ for $1 \leq k \leq d$.

For the purpose of formulating loop transformations, we consider D to be a subset of the d-dimensional vector space over rationals. Throughout the paper, we let $I = (i_1, \ldots, i_d)$ and $J = (j_1, \ldots, j_d)$. With the domain and tuple notations, Loop Nest 1 can be rewritten as follows:

Loop Nest 2

$$\begin{array}{l}\mathsf{DO}\ (I{:}D)\,\{\\ \quad body\,\}\end{array}$$

In this paper, we focus on sequential loop nests which are perfectly nested. We use the following loop nest as a generic example throughout the paper, where D is a d dimensional index domain and $\tau[a]$ is an expression containing a:

Loop Nest L (Generic Loop Nest)

$$\begin{array}{l}\mathsf{DO}\ (I{:}D)\,\{\\ \quad \ldots\\ \quad S_1:\ \mathsf{IF}(P_1)\ A(X(I)) = \ldots\\ \quad \ldots\\ \quad S_2:\ \mathsf{IF}(P_2)\ B(Z(I)) = \tau[A(Y(I))]\\ \quad \ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}\end{array}$$

Data Dependence We now define dependence between statements. Let S_1 and S_2 be two statements of a program. A *flow dependence* exists from S_1 to S_2 if S_1 writes data that can subsequently be read by S_2. An *anti-dependence* exists from S_1 to S_2 if S_1 reads data that S_2 can subsequently overwrite. An *output dependence* exists from S_1 to S_2 if S_1 writes data that S_2 can subsequently overwrite. We use the notation $S_1 \Rightarrow S_2$ to denote a dependence from S_1 to S_2.

Equivalence Classes over Statements in a Loop Nest Let "$\stackrel{*}{\Rightarrow}$" be the reflexive and transitive closure of the dependence relation "$\Rightarrow$" over statements. We define a binary operation "$\sim$" over statements where $S_1 \sim S_2$ if $S_1 \stackrel{*}{\Rightarrow} S_2$ and $S_2 \stackrel{*}{\Rightarrow} S_1$. Note that "$\sim$" is an equivalence relation, and therefore, partitions loop statements into equivalence classes (called π blocks in [34]). The technique of *loop fission* [34] can be applied to spilt the loop nest into several new loop nests, one for each equivalence class.

Dependent versus Independent Blocks By the definition of the equivalence relation "$\sim$", a single statement which is not self-dependent can form an equivalence class on its own. This case must be distinguished from all others where cyclic dependences actually occur. We call this special case of an equivalence class under the dependence relation an *independent* block, and others *dependent* blocks. For example, consider the following loop nest:

Loop Nest 3

$$
\begin{array}{l}
\mathsf{DO}\ (i = 1, n)\,\{ \\
\quad \mathsf{DO}\ (j = i + 1, n)\,\{ \\
\qquad S_1 :\ A(i, j - i) = B(i, j - i - 1) \\
\qquad S_2 :\ B(i, j - i) = A(i - 1, j - i) \\
\qquad S_3 :\ C(i, j) = A(i, j) + B(i - 1, j + 1)\ \}\ \}
\end{array}
$$

Statements S_1 and S_2 are in the same dependent block because of cyclic flow dependences, and statement S_3 forms an independent block. Loop fission can be applied to split Loop Nest 3 into two new loop nests:

Loop Nest 4

$$
\begin{array}{l}
\mathsf{DO}\ (i = 1, n)\,\{ \\
\quad \mathsf{DO}\ (j = i + 1, n)\,\{ \\
\qquad S_1 :\ A(i, j - i) = B(i, j - i - 1) \\
\qquad S_2 :\ B(i, j - i) = A(i - 1, j - i)\ \}\ \} \\
\mathsf{DO}\ (i = 1, n)\,\{ \\
\quad \mathsf{DO}\ (j = i + 1, n)\,\{ \\
\qquad S_3 :\ C(i, j) = A(i, j) + B(i - 1, j + 1)\ \}\ \}
\end{array}
$$

Loop fission can be used to separate an independent block from other dependent blocks, and the new loop nest consisting of one independent block is readily parallelizable. Similarly, loop fission can be used to transform a loop body containing multiple dependent blocks into multiple loop nests, each with a single dependent block. We therefore consider the case that Loop Nest L consists of one dependent block for the rest of the paper. A loop nest consisting of a dependent block may be parallelized by several techniques, namely statement reordering, loop vectorization, interchanging and permutation [7, 34]. To determine whether these transformations are applicable or not, the notion of a direction vector [34] is necessary.

Direction Vectors Consider Loop Nest L. For statement S_2 to compute the value $B(Z(J))$ at iteration J, the value $A(Y(J))$ is needed. If $A(Y(J))$ is computed from statement S_1 at iteration I, i.e. $Y(J) = X(I)$, then we say S_2 at iteration J is flow dependent on S_1 at iteration I, denoted by $S_1@I \Rightarrow S_2@J$.

For a dependence $S_1@I \Rightarrow S_2@J$, the vector $(\mathbf{sig}(i_1 - j_1), \ldots, \mathbf{sig}(i_d - j_d))$ is called a *direction vector* from S_1 to S_2 [34], where **sig** is a function from the set of integers to the set of ordering relations "<", "=", and ">":

$$
\mathbf{sig}(z) = \left\{ \begin{array}{l} z < 0 \rightarrow \text{"<"} \\ z = 0 \rightarrow \text{"="} \\ z > 0 \rightarrow \text{">"} \end{array} \right\} \tag{1}
$$

Dependence Vectors Loop skewing is another transformation which may expose more parallelism, if the parallelism gained from the above-mentioned techniques is insufficient. In order to do loop skewing, we need to know the relative positions of index tuples I and J for each dependence $S_1@I \Rightarrow S_2@J$ [25, 34]. For a dependence $S_1@I \Rightarrow S_2@J$, the vector $(j_1 - i_1, \ldots, j_d - i_d)$ is called a *dependence vector* from S_1 to S_2 [34].

Since I and I are in the d-dimensional vector space, we use $I+J$ to denote the addition of two vectors I and J, i.e. $I+J = (i_1+j_1, \ldots, i_d+j_d)$; and similarly, $J-I = (j_1-i_1, \ldots, j_d-i_d)$.

Notation for Concatenation Since we will be using matrix and vector notations extensively, we define the notation for matrix concatenation here. We treat a row vector of length d as a degenerate 1-by-d matrix, a column vector of length d as a degenerate d-by-1 matrix, and a scalar as a degenerate 1-by-1 matrix.

A *horizontal concatenation* of an l-by-m matrix A and an l-by-n matrix B, denoted by $[A, B]$, is an l-by-$(m+n)$ matrix, where the (i,j)-th element of $[A, B]$ is equal to the (i,j)-th element of A if $j \leq m$, or it is equal to the $(i, j-m)$-th element of B if $j > m$.

A *vertical concatenation* of an m-by-l matrix A and an n-by-l matrix B, denoted by $\begin{bmatrix} A \\ B \end{bmatrix}$, is an $(m+n)$-by-l matrix, where the (i,j)-th element of $\begin{bmatrix} A \\ B \end{bmatrix}$ is equal to the (i,j)-th element of A if $i \leq m$, or it is equal to the $(i-m, j)$-th element of B if $i > m$.

3 Formalizing Loop Transformation

We now formalize the notion of loop transformation from a source loop nest to a target parallel loop nest. A *loop transformer* is a function defined over the Cartesian product of the iteration space of the loop nest and the set of statements in the body of the loop that relates a given loop nest with its transformed version. From the standpoint of symbolic transformation of the program text, a loop transformer can be decomposed into two components: the first component, called *domain morphism*, defines how the iteration space should be mapped to a new one (with new loop bounds and possibly new predicates guarding the loop body), and the second component, called *statement reordering function*, defines the ordering of the statements in the transformed loop nest. The process of obtaining a loop transformer, however, suggests another decomposition: a *temporal morphism* and a *spatial morphism.*

3.1 Loop Transformer and Schedule

Kinds of Index Domains For the purpose of loop transformation, it is useful to indicate how the index domain shall be interpreted. We do this by defining *kinds* of index domains. The kind of an interval domain D can be either *spatial* or *temporal.* The kind of a product domain is the product of the kinds of the component domains. For example, $D_1 \times D_2$ is of kind temporal×spatial if D_1 is of kind temporal and D_2 is of kind spatial. A single-level

loop with a temporal index domain corresponds to a sequential loop (i.e. DO), while a spatial index domain corresponds to a parallel loop (i.e. DOALL).

Lexicographical Ordering We use the following notations to denote lexicographical ordering on elements X and Y of an n-dimensional index domain. We define "$\prec$" to be the lexicographical ordering: we say $X \prec Y$ if there exists k, $1 \leq k \leq n$, such that $x_l = y_l$ for all l, $l < k$, and $x_k < y_k$. Similarly, we say $X \preceq Y$ if $X \prec Y$ or $x_k = y_k$ for all k, $1 \leq k \leq n$. We use $\hat{0}$ to denote the zero vector.

Domain Morphism We define a *domain morphism* to be a bijective function g from index domain D to index domain E, denoted by $g{:}D \rightarrow E$, such that for all dependences $S_1@I \Rightarrow S_2@J$, condition $g(J) - g(I) \succeq \hat{0}$ holds. In other words, a domain morphism will never reverse the ordering imposed by dependence relations.

In this paper, we restrict the codomain E of a domain morphism to be a cross product of a temporal index domain E_1 and a spatial index domain E_2, i.e. $E = E_1 \times E_2$. Under this restriction, all parallel loops are innermost loops in the transformed loop nest. We define g_1 and g_2 to be two functions:

$$g_1 : D \rightarrow E_1, \quad \text{(called a } \textit{temporal morphism}\text{) and} \tag{2}$$

$$g_2 : D \rightarrow E_2. \quad \text{(called a } \textit{spatial morphism}\text{)} \tag{3}$$

Under domain morphism g, index I in the original loop will be mapped to index $J = g(I)$ in the transformed loop nest. Since g is bijective, it has a well-defined inverse, denoted by g^{-1}. Clearly, $I = g^{-1}(J)$. The following loop nest

Loop Nest 5

```
DO ((I:D)) {
    ... A(X(I)) ... }
```

will be transformed into the following new loop nest under domain morphism $g{:}D \rightarrow E_1 \times E_2$:

Loop Nest 6

```
DO ((J1:E1)) {
    DOALL ((J2:E2)) {
        ... A(X(g^-1 [J1; J2])) ... } }
```

where $\begin{bmatrix} J_1 \\ J_2 \end{bmatrix}$ denotes the vertical concatenation of two column vectors J_1 and J_2.

The requirement of g to be surjective is in fact not essential. For any injective function $g':D \rightarrow E$, we can always derive a corresponding bijective function $g:D \rightarrow \{g'(I) \mid I \in D\}$ from D to the image of D under g' [11]. Therefore, by allowing the codomain of a bijective function to be the image of an injective function, we allow a much more general class of functions to be used as domain morphism. For comparison, the unimodular transformations discussed in [8, 33] are special classes of bijective functions. The generality does require some nontrivial algebraic manipulation to generate correct loop bounds and predicates to guard the conditional statements in the transformed loop nest. An automatic transformation procedure for doing this based on an equational theory is described in [11].

Statement Reordering We now discuss statement reordering. Let $\mathcal{S}$ denote the set of statements in the loop body. We define a *statement reordering* to be a function h from the set of statements to the set of statement labels:

$$r:\mathcal{S} \rightarrow [0, s-1], \text{ where } s = |\mathcal{S}|, \text{ the number of statements in } \mathcal{S}. \tag{4}$$

Loop Transformer With g and r defined above, the following function h, called the *loop transformer*, specifies how a loop nest is transformed:

$$\begin{aligned} &h:D \times \mathcal{S} \rightarrow E_1 \times E_2 \times [0, s-1] \\ &h(I, S) = (g_1(I), g_2(I), r(S)). \end{aligned} \tag{5}$$

Schedule Given h defined above, a *schedule* π is defined to be a function

$$\begin{aligned} &\pi:D \times \mathcal{S} \rightarrow E_1 \times [0, s-1] \\ &\pi(I, S) = (g_1(I), r(S)), \end{aligned} \tag{6}$$

such that condition $\pi(J, S_2) - \pi(I, S_1) \succ \hat{0}$ must hold for all dependences $S_1@I \Rightarrow S_2@J$ in the loop nest. The condition ensures that the ordering imposed by dependence relations is preserved. Clearly, a schedule determines the sequential execution of the transformed parallel loop nest. Note that by the definition of domain morphism, $g_1(J) - g_1(I)$ can be equal to the zero vector, i.e. $S_1@I$ and $S_2@J$ can be computed at the same iteration in the transformed loop nest. In this case, statement S_1 must be in front of statement S_2 in the loop body, i.e. condition $r(S_1) < r(S_2)$ must hold, to preserve the dependence ordering.

3.2 Overall Procedure to Obtain a New Loop Nest

Finding a schedule π is to understand what is the potential parallelism that can be extracted from the source program. There may be alternative schedules which are incomparable without a target machine model. Traditional loop transformation uses an ad hoc approach in choosing a particular schedule out of several alternative ones. A systematic, cost-driven approach to choose alternative schedules is beyond the scope of this paper. This paper gives algorithms to find schedules which result in maximal parallelism of the innermost loops.

The so-called *strip mining* [34] and *tiling* [33, 36] of loops are captured by the spatial morphism g_2. The choice of g_2, which depends on factors such as memory and processor organization and communication cost, can be dealt with separately and is not included in this paper. However, given a schedule $\pi = (g_1, r)$, a valid g_2 should keep a loop transformer $h = (g_1, g_2, r)$ injective. In this paper, we use the following default spatial morphism for all the examples: $g_2(i_1, \ldots, i_d) = (i_{p_1}, \ldots, i_{p_n})$, so as to result in a loop transformer h that is injective, where n is the dimensionality of the spatial index domain E_2, $\{p_1, \ldots, p_n\}$ is a subset of interval domain $[1, d]$, and $p_1 \prec \ldots < p_n$.

Overall Procedure To summarize, the overall procedure to obtain a new loop nest is:

1. First generate a schedule $\pi = (g_1, r)$ to maximize the degree of parallelism.
2. Then determine the spatial morphism g_2 of domain morphism based on target machine characteristics such as memory and processor organization, communication cost, etc., or use a default function as shown above.
3. The loop transformer is simply $h = (g_1, g_2, r)$.
4. Finally perform symbolic program transformation, given the source loop nest and loop transformer h, to obtain the new loop nest. For the formal procedure, please refer to [11].

The remainder of this paper is devoted to generating π to gain large scale parallelism.

4 Classes of Affine and Piece-Wise Affine Schedules

We call a schedule affine if it is an affine function of the loop indices. We call a schedule piece-wise affine if the restriction of the function to each subdomain of D and each subset of S is affine. In the loop restructuring literature, only affine schedules are considered. In this paper, we consider, in addition, piece-wise affine schedules.

In order to discuss the algorithms for generating suitable schedules, we now classify them according to two properties: (1) the uniformity of the schedule with respect to the the set of statements S and the index domain D, and (2) the degree of parallelism in the transformed Loop Nest.

4.1 Properties of Schedules

Uniformity Let index domain D be partitioned into m disjoint subdomains D_k, $1 \leq k \leq m$; and let the set of statements S be partitioned into n disjoint subsets S_k, $1 \leq k \leq n$. The general form of a piece-wise affine schedule π defined in Equation (6) consists of conditional branches, one for each pair of subdomain D_i and statement subset S_j, and an affine expression of the loop indices is on the right-hand side of each branch. We call a schedule

1. *uniform* if $m = 1$ and $n = 1$,
2. *subdomain-variant* if $m > 1$ and $n = 1$, (also called a subdomain schedule)
3. *statement-variant* if $m = 1$ and $n > 1$, or
4. *nonuniform* if $m > 1$ and $n > 1$.

Degree of Generated Parallelism As defined in Equations (3) and (6), the dimensionality of E_1, the temporal index domain, indicates the number of levels of sequential loops in the transformed loop nest. Hence a schedule π would generate a target loop nest with more levels of parallel loops and thus potentially more parallelism if E_1 is of lower dimensionality. We call the dimensionality of E_1 the *sequential level* of π. Schedules can thus be classified as:

1. *Single-sequential level schedule* if E_1 is a subset of the set of natural numbers $\mathcal{N}$.
2. *Multiple-sequential level schedule* if E_1 is a subset of $\mathcal{N}^n$, where n is a positive integer and $n \leq d$, the dimensionality of the original loop nest.
3. *Mixed schedule* if E_1 can be of different dimensions for each pair of subdomain D_i and statement subset S_j. Such a mixed schedule will result in transformed programs consisting of imperfectly nested loops.

4.2 Classification and Functional Form of Schedules

Classification Clearly, the uniformity of π and the dimensionality of π are two orthogonal properties, except that a mixed schedule cannot be uniform. Thus there are all together eleven $(4 * 3 - 1)$ classes of affine and piece-wise affine schedules. The classes and their acronyms ranging from single-sequential level uniform schedules to mixed nonuniform schedules are given below:

	Single-Sequential Level (SSL)	Multiple-Sequential Level (MSL)	Mixed
Uniform (U)	SSL-U	MSL-U	
Subdomain (SD)	SSL-SD	MSL-SD	Mixed-SD
Statement-Variant (SV)	SSL-SV	MSL-SV	Mixed-SV
Nonuniform (NU)	SSL-NU	MSL-NU	Mixed-NU

Functional Form We now describe the forms of affine and piece-wise affine schedules by using matrix and vector notations. Let $r(S)$ for a given S in $\mathcal{S}$ be a constant scalar. Let d be the dimensionality of the index domain of the source loop nest.

Uniform Schedule:

$$\pi(I,S) = (TI, r(S)), \quad I \in D, S \in \mathcal{S}, \tag{7}$$

where T is a constant l-by-d matrix and l is the sequential level of the schedule π.

Subdomain Schedule:

$$\pi(I,S) = \left\{ \begin{array}{l} I \in D_1 \rightarrow (T_1 I, r_1(S)) \\ \ldots \\ I \in D_m \rightarrow (T_m I, r_m(S)) \end{array} \right\}, \quad I \in D, S \in \mathcal{S}, \tag{8}$$

where T_i, $1 \leq i \leq m$, is a constant l_i-by-d matrix and l_i is the sequential level of the part of the schedule defined over D_i.

Statement-Variant Schedule:

$$\pi(I,S) = \left\{ \begin{array}{l} S \in \mathcal{S}_1 \rightarrow (T_1 I, r(S)) \\ \ldots \\ S \in \mathcal{S}_n \rightarrow (T_n I, r(S)) \end{array} \right\}, \quad I \in D, S \in \mathcal{S}, \tag{9}$$

where T_i, $1 \leq i \leq n$, is a constant l_i-by-d matrix and l_i is the sequential level of the part of the schedule defined over $\mathcal{S}_i$.

Nonuniform Schedule:

$$\pi(I,S) = \left\{ \begin{array}{l} (I \in D_1) \rightarrow \left\{ \begin{array}{l} (S \in \mathcal{S}_1) \rightarrow (T_{11} I, r_1(S)) \\ \ldots \\ (S \in \mathcal{S}_n) \rightarrow (T_{1n} I, r_1(S)) \end{array} \right\} \\ \ldots \\ (I \in D_m) \rightarrow \left\{ \begin{array}{l} (S \in \mathcal{S}_1) \rightarrow (T_{m1} I, r_m(S)) \\ \ldots \\ (S \in \mathcal{S}_n) \rightarrow (T_{mn} I, r_m(S)) \end{array} \right\} \end{array} \right\}, \quad I \in D, S \in \mathcal{S}, \tag{10}$$

where T_{ij}, $1 \leq i \leq m$ and $1 \leq j \leq n$, is a constant l_{ij}-by-d matrix and l_{ij} is the sequential level of the part of the schedule defined over D_i and $\mathcal{S}_j$.

The linear term TI, $I \in D$, determines the form of the sequential loops in the transformed loop nest, which includes nesting structures, bounds, and possibly additional predicates to guard the loop body. The constant terms $r(S)$ determine the orders of the statements in the transformed loop body.

4.3 Examples of Different Classes of Schedules

We now give some examples of different classes of schedules. We first show that loop vectorization, interchanging, permutation and skewing are special cases of multiple-sequential uniform schedules.

Example 1: Loop Vectorization Let loc be a function from $\mathcal{S}$ to $\mathcal{N}$ that returns the position of the statement S in the source loop nest. Suppose a dependence test says that m innermost loops can be parallelized. The schedule for the so-called loop vectorization of the $d-m$ outermost loops is of the following form, where d is the dimensionality of the index domain of the loop nest:

$$\pi(I,S) = (i_1, i_2, \ldots, i_{d-m}, \mathsf{loc}(S)), \tag{11}$$

$$\text{i.e.} \qquad T = \begin{pmatrix} V(1) \\ \ldots \\ V(d-m) \end{pmatrix}, \text{ and} \tag{12}$$

$$r(S) = \mathsf{loc}(S), \tag{13}$$

where each $V(k)$ is a vector of length d with k-th element being 1 and all other elements being 0.

Example 2: Loop Interchanging and Permutation Loop interchanging and loop permutation [1, 2, 3, 7, 34, 35, 37] is a process of switching inner and outer loops. Suppose Loop Nest 1 after loop interchanging or loop permutation becomes

$$\begin{array}{l} \mathsf{DO}\ (i_{p_1} = l_{p_1}, u_{p_1})\,\{ \\ \quad \mathsf{DO}\ (\ldots)\,\{ \\ \quad\quad \mathsf{DO}\ (i_{p_d} = l_{p_d}, u_{p_d})\,\{ \\ \quad\quad\quad \mathsf{body}\ \}\qquad\qquad \}\ \}, \end{array}$$

where $(p_1, p_2, \ldots, p_d)$ is a permutation of $(1, 2, \ldots, d)$. Also suppose the m innermost loops are parallelizable. The schedule π has the form:

$$\pi(I,S) = (i_{p_1}, i_{p_2}, \ldots, i_{p_{d-m}}, \mathsf{loc}(S)), \tag{14}$$

$$\text{i.e.} \qquad T = \begin{pmatrix} V(p_1) \\ \ldots \\ V(p_{d-m}) \end{pmatrix}, \text{ and} \tag{15}$$

$$r(S) = \mathsf{loc}(S), \tag{16}$$

where each $V(k)$ is the same as defined in Example 1.

Example 3: Loop Skewing This operation transforms Loop Nest 1 as follows: shifting index i_n with respect to index i_m, $1 \leq m < n \leq d$, by a factor of f, where f is a positive integer, replacing l_n with the expression $(l_n + i_m * f)$, replacing u_n with the expression $(u_n + i_m * f)$, and replacing all occurrences of i_n in the loop with the expression $(i_n - i_m * f)$ [34, 37]. The transformed loop nest is of the form:

Loop Nest 7

DO $(i_1 = l_1, u_1)\{$

$\ldots$

DO $(i_n = l_n + i_m * f, u_n + i_m * f)\{$

$\ldots$

DO $(i_d = l_d, u_d)\{$

same loop body but with i_n being replaced by $(i_n - i_m * f)$ $\}\}\}$

The schedule for such so called loop skewing is of the form:

$$\pi(I,S) = (i_1, \ldots, i_m, \ldots, \underbrace{i_n + f * i_m}_{n\text{-th element}}, \ldots, i_d, \mathsf{loc}(S)), \tag{17}$$

i.e. $$T = \begin{pmatrix} V(1) \\ \cdots \\ V(n) + f * V(m) \\ \cdots \\ V(d) \end{pmatrix}, \text{ and} \tag{18}$$

$$r(S) = \mathsf{loc}(S), \tag{19}$$

where each $V(k)$ is the same as defined in Example 1.

Example 4: Single-Sequential Level Uniform Schedule

Loop Nest 8

DO $(i = 1, n)\{$

DO $(j = 1, n)\{$

$S_1 : A(i,j) = B(i,j-1) + i$

$S_2 : B(i,j) = A(i-1,j) + j\ \}\}$

A single-sequential level uniform schedule

$$\pi((i,j,k), S_1) = (i, 1), \quad \text{and} \tag{20}$$

$$\pi((i,j,k), S_2) = (i, 0), \tag{21}$$

will transform Loop Nest 8 into

Loop Nest 9

$$
\begin{array}{l}
\mathsf{DO}\ (i = 1, n)\,\{ \\
\quad \mathsf{DOALL}\ (j = 1, n)\,\{ \\
\quad\quad S_2 :\ B(i,j) = A(i-1,j) + j \\
\quad\quad S_1 :\ A(i,j) = B(i,j-1) + i\ \}\ \}
\end{array}
$$

Example 5: Multiple-Sequential Level Uniform Schedule

Loop Nest 10

$$
\begin{array}{l}
\mathsf{DO}\ (i = n-1, 1, -1)\,\{ \\
\quad \mathsf{DO}\ (j = i+1, n)\,\{ \\
\quad\quad \mathsf{DO}\ (k = i, j)\,\{ \\
\quad\quad\quad S_1 :\ \mathsf{IF}(i+1 = k)\ B(i,j,k) = C(i+1,j,j) \\
\quad\quad\quad S_2 :\ \mathsf{IF}(i+1 < k)\ B(i,j,k) = B(i+1,j,k) \\
\quad\quad\quad S_3 :\ \mathsf{IF}(i+j+1 < 2k)\ C(i,j,k) = C(i,j,k-1) + B(i,j,k)\ \}\ \}\ \}
\end{array}
$$

A two-sequential level uniform schedule

$$\pi((i,j,k), S) = ((-i,k), \mathsf{loc}(S)) \tag{22}$$

will transform Loop Nest 10 into

Loop Nest 11

$$
\begin{array}{l}
\mathsf{DO}\ (i = 1-n, -1)\,\{ \\
\quad \mathsf{DO}\ (k = -i, n)\,\{ \\
\quad\quad \mathsf{DOALL}\ (j = 1-i, n)\,\{ \\
\quad\quad\quad S_1 :\ \mathsf{IF}((-i+1 = k) \wedge (k \le j))\ B(-i,j,k) = C(1-i,j,j) \\
\quad\quad\quad S_2 :\ \mathsf{IF}((-i+1 < k) \wedge (k \le j))\ B(-i,j,k) = B(1-i,j,k) \\
\quad\quad\quad S_3 :\ \mathsf{IF}((-i+j+1 < 2k) \wedge (k \le j)) \\
\quad\quad\quad\quad C(-i,j,k) = C(-i,j,k-1) + B(-i,j,k) \qquad\qquad \}\ \}\ \}
\end{array}
$$

Example 6: Mixed Statement-Variant Schedule Consider Loop Nest 10 again. The following schedule

$$\pi((i,j,k), S) = \left\{ \begin{array}{l} S = S_3 \rightarrow ((-i,k), \mathsf{loc}(S)) \\ \mathsf{else} \rightarrow (-i, \mathsf{loc}(S)) \end{array} \right\} \tag{23}$$

transforms Loop Nest 10 to Loop Nest 12 below, which consists of imperfectly nested loops:

Loop Nest 12

$$
\begin{array}{l}
\mathsf{DO}\ (i = 1-n, -1)\,\{ \\
\quad \mathsf{DOALL}\ ((j = 1-i, n), (k = -i, n))\,\{ \\
\quad\quad S_1 :\ \mathsf{IF}((-i+1 = k) \wedge (k \leq j))\ B(-i, j, k) = C(1-i, j, j) \\
\quad\quad S_2 :\ \mathsf{IF}((-i+1 < k) \wedge (k \leq j))\ B(-i, j, k) = B(1-i, j, k)\,\} \\
\quad \mathsf{DO}\ (k = -i, n)\,\{ \\
\quad\quad \mathsf{DOALL}\ (j = 1-i, n)\,\{ \\
\quad\quad\quad S_3 :\ \mathsf{IF}((-i+j+1 < 2k) \wedge (k \leq j)) \\
\quad\quad\quad\quad C(-i, j, k) = C(-i, j, k-1) + B(-i, j, k)\,\}\,\} \qquad\qquad \}
\end{array}
$$

Example 7: Single-Sequential Level Subdomain-Variant Schedule Another possible transformation of Loop Nest 10 is the schedule:

$$\pi((i,j,k),S) = \left\{ \begin{array}{l} i+j-2k < 0 \rightarrow (-2i+j+k, \mathsf{loc}(S)) \\ i+j-2k \geq 0 \rightarrow (-i+2j-k, \mathsf{loc}(S)) \end{array} \right\}, \tag{24}$$

which transforms Loop Nest 10 into:

Loop Nest 13

$$
\begin{array}{l}
\mathsf{DO}\ (t = 2, 2n-2)\,\{ \\
\quad \mathsf{DOALL}\ (i = n-1, 1, -1)\,\{ \\
\quad\quad \mathsf{DOALL}\ (j = i+1, n)\,\{ \\
\quad\quad\quad S_{11} :\ \mathsf{IF}((2t+3i-3j > 0) \wedge (t+i-j-1 = 0)) \\
\quad\quad\quad\quad B(i, j, t+2i-j) = C(i+1, j, j) \\
\quad\quad\quad S_{12} :\ \mathsf{IF}((2t+3i-3j \geq 0) \wedge (t+2i-2j+1 = 0)) \\
\quad\quad\quad\quad B(i, j, -t-i+2j) = C(i+1, j, j) \\
\quad\quad\quad S_{21} :\ \mathsf{IF}((2t+3i-3j > 0) \wedge (t+i-j-1 > 0)) \\
\quad\quad\quad\quad B(i, j, t+2i-j) = B(i+1, j, t+2i-j) \\
\quad\quad\quad S_{22} :\ \mathsf{IF}((2t+3i-3j \geq 0) \wedge (t+2i-2j+1 < 0)) \\
\quad\quad\quad\quad B(i, j, -t-i+2j) = B(i+1, j, -t-2i+2j) \\
\quad\quad\quad S_{31} :\ \mathsf{IF}((2t+3i-3j > 0) \wedge (2t+3i-3j-1 > 0)) \\
\quad\quad\quad\quad C(i, j, t+2i-j) = C(i, j, t+2i-j-1) + B(i, j, t+2i-j) \\
\quad\quad\quad S_{32} :\ \mathsf{IF}((2t+3i-3j \geq 0) \wedge (2t+3i-3j+1 < 0)) \\
\quad\quad\quad\quad C(i, j, -t-i+2j) = C(i, j, -t-i+2j-1) + B(i, j, -t, -i+2j)\,\}\,\}\,\}
\end{array}
$$

Since there are two affine functions for disjoint subdomains of the index domain of the loop nest, each statement in Loop Nest 10 results in two guarded statements in the transformed

loop nest. In fact Loop Nest 10 is part of the dynamic programming code presented in Section 8. As one can see, an SSL-SD schedule can result in code of considerable complexity. It would be a very tedious and error-prone process for a user to write the code by hand. But a compiler can generate the new loop nest, given the schedule, and the original loop nest mechanically.

5 Algorithms for Generating Single-Sequential Level Schedules

Our algorithms for generating the various single-sequential level schedules are based on Quinton's algorithm for generating SSL-U schedules [13, 29, 30]. To make the paper self-contained, we first review Quinton's algorithm. We then present two algorithms, one for generating SSL-SD schedules and the other for SSL-SV schedules.

5.1 Previous Work: Uniform Scheduling Algorithm

Quinton's approach addresses the analysis and mapping of linear recurrence equations [13, 29, 30]. We formulate Quinton's algorithm in the context of loop transformations.

5.1.1 Problem Formulation

Constraints Derived from Data Dependences For an SSL-U schedule $\pi(I, S) = (TI, r(S))$, where T is a row vector, the inequality

$$(TJ, r(S_2)) - (TI, r(S_1)) \succ \hat{0}, \tag{25}$$

must hold for all index tuples I and J in index domain D and statements S_1 and S_2 in $\mathcal{S}$ such that $S_1@I \Rightarrow S_2@J$. We first focus on the problem of obtaining T satisfying the following more stringent condition:

$$TJ - TI = T(J - I) > 0 \tag{26}$$

for each dependence $S_1@I \Rightarrow S_2@J$. If such T exists, then Equation (25) also holds for all dependences $S_1@I \Rightarrow S_2@J$ due to the lexicographical ordering "$\succ$" regardless of what r is. In this case, the ordering among statements can be arbitrary. How to obtain T satisfying less stringent conditions, and use r, in addition to T, to preserve the ordering imposed by dependences, will be discussed in Section 5.2.

Space of Dependence Vectors It is clear that in the case of uniform schedule, dependence vectors $J - I$ are sufficient for obtaining T. We now formulate the set of dependence vectors for conditional statements.

Let D be an index domain and P be the predicate in a conditional statement S. We define the *index domain of statement S* to be D under the restriction of P, denoted by $D{\downarrow}P$:

$$D{\downarrow}P = \{I \mid I \in D, \text{and } P(I) \text{ is true}\}. \tag{27}$$

For the rest of the paper, we consider only flow dependence in Loop Nest L. Anti-dependence and output dependence can be treated similarly. In this case, the set of dependence vectors from statement S_1 to S_2 in Loop Nest L, denoted by $\mathcal{V}(S_1, S_2)$, is [23]:

$$\mathcal{V}(S_1, S_2) = \{J - I \mid I \in (D{\downarrow}P_1), J \in (D{\downarrow}P_2), \text{ and } X(I) = Y(J)\}. \tag{28}$$

Input System As described before, each dependence relation $S_1@I \Rightarrow S_2@J$ defines an inequality $T(J - I) > 0$ that row vector T must satisfy. We call the set of all constraints on T the *input system*, denoted by $\mathcal{C}$:

$$\begin{aligned} \mathcal{C} &= \{T(J-I) > 0 \mid \text{there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \text{ and index tuples} \\ &\qquad I \text{ and } J \text{ in } D, \text{ such that } S_1@I \Rightarrow S_2@J\} \\ &= \{T(J-I) > 0 \mid \text{there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \text{ such that} \\ &\qquad (J-I) \in \mathcal{V}(S_1, S_2)\}. \end{aligned} \tag{29}$$

Polyhedra and Polytopes There can be many dependence vectors that need to be considered, and they can be infinitely many when the loop bounds are unknown at compile time. We need to rely on a technique that decomposes a polyhedron into vertices and extremal rays [13, 30, 32] to manage the complexity of the algorithm. We first define what a polyhedron is.

Let A be a c-by-d matrix and B be a column vector of length d. Let $\mathcal{Z}$ be the set of rationals. A *polyhedron* is a subspace of the d-dimensional vector space $\mathcal{Z}^d$ that can be expressed as $\{I \mid AI \geq B\}$ [32]. A bounded polyhedron is called a *polytope* [32].

Constraints on the Input System In the loop restructuring literature, only *rectangular* and *trapezoid* index domains of loop nests are considered [6, 34]. Note that rectangular and trapezoid index domains are special classes of polyhedra [23]. In order to obtain uniform schedules systematically, Quinton restricts the index domain of each recurrence equation to be a polyhedron and all subscript functions (called *index mappings* in [30]) to be affine expressions of the indices used in defining the index domains. In Fortran like programs, the above restriction is translated to perfectly nested loops where loop bounds at one level may depend on the outer levels, with the loop body consisting of conditional statements where all the predicates of conditionals and all subscript functions are affine expressions of the loop indices. Under these restrictions, Quinton shows that [30] the set $\mathcal{V}(S_1, S_2)$ of the dependence vectors from statement S_1 to S_2 is a d-dimensional polyhedron, where d is the dimensionality of the index domain of the loop nest.

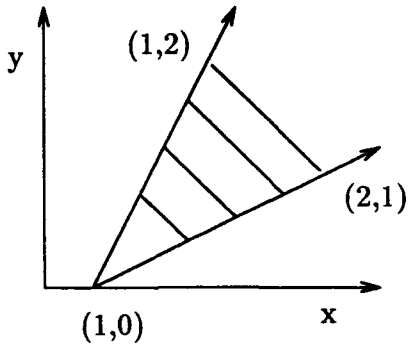

Figure 1: Point (1,0) is the vertex and two vectors (1,2) and (2,1) are the extremal rays.

5.1.2 Decomposing a Polyhedron into Vertices and Extremal Rays

We now discuss how to represent polyhedron $\mathcal{V}(S_1, S_2)$ by its vertices and extremal rays.

Vertices and Extremal Rays Any polyhedron can be decomposed into a finite set of *vertices* and *extremal rays* [32]. (Since a *line* can be interpreted as two rays in opposite directions [32], vertices and extremal rays are sufficient for polyhedra decomposition.) Here, we use the following example to show what vertices and extremal rays are. For formal definitions, please refer to [32]. Consider the polyhedron P_e specified by two inequalities: $P_e = \{(x, y) \mid 2x - y \geq 2, \text{ and } 2y - x \geq -1\}$. Point $(1, 0)$ is the vertex and two vectors $(1, 2)$ and $(2, 1)$ are the extremal rays of P_e as shown below:

Algorithm for Polyhedron Decomposition Since the algorithm for obtaining subdomain schedules (to be discussed in Section 5.3) relies on the algorithm for decomposing a polyhedron into vertices and rays, we now review the decomposition algorithm presented in [13].

Let Q be a polyhedron defined by $Q = \{K \mid AK \geq B\}$, where A is a c-by-d matrix and B is a column vector of length c (d is the dimensionality of index domain D and c is the number of constraints). If B is a zero vector, then we call Q a *homogeneous* polyhedron, otherwise, a *nonhomogeneous* polyhedron. A nonhomogeneous polyhedron has a corresponding homogeneous polyhedron. Let h be a scalar variable. Let $Q_h = \{\begin{bmatrix} K \\ h \end{bmatrix} \mid [A, -B]\begin{bmatrix} K \\ h \end{bmatrix} \geq 0, h \geq 0\}$, where $\begin{bmatrix} K \\ h \end{bmatrix}$ is the vertical concatenation of column vector K and scalar h, and $[A, -B]$ is the horizontal concatenation of matrix A and column vector

B. Then Q_h defines the corresponding homogeneous polyhedron of Q. A ray $\begin{bmatrix} R \\ h \end{bmatrix}$ of Q_h implies that R is a ray of Q if $h = 0$, or $\frac{R}{h}$ is a vertex of Q if $h > 0$. Therefore, it suffices to compute the extremal rays of a homogeneous polyhedron, and then obtain the vertices and extremal rays for the corresponding nonhomogeneous polyhedron.

Algorithm ER (Obtaining Extremal Rays of a Homogeneous Polyhedron)
The algorithm starts with an initial set $\mathcal{R}$ of rays which can be easily generated. A succession of transformations on $\mathcal{R}$ are then performed, one for each constraint in the system. The ordering in which the constraints are chosen does not affect the correctness of the result. In each transformation, new $\mathcal{R}$ is computed according to the projections of the current rays in $\mathcal{R}$ on C as described below, where $C \begin{bmatrix} K \\ h \end{bmatrix} \geq 0$ is the current selected constraint:

1. $\mathcal{R}_0 \leftarrow \{R \mid R \in \mathcal{R}, CR = 0\}$;
2. $\mathcal{R}_+ \leftarrow \{R \mid R \in \mathcal{R}, CR > 0\}$;
3. $\mathcal{R}_- \leftarrow \{R \mid R \in \mathcal{R}, CR < 0\}$;
4. $\mathcal{R} \leftarrow \mathcal{R}_0 \cup \mathcal{R}_+ \cup \{(\alpha_1 R_1 + \alpha_2 R_2) \mid R_1 \in \mathcal{R}_+, R_2 \in \mathcal{R}_-, \alpha_1 > 0, \alpha_2 > 0, C(\alpha_1 R_1 + \alpha_2 R_2) = 0\}$.

5.1.3 Linear Programming Formulation

After polyhedron decomposition, each point in polyhedron $\mathcal{V}(S_1, S_2)$ can be expressed as the sum of a convex combination of the vertices and of a positive combination of the extremal rays of $\mathcal{V}(S_1, S_2)$ [32]. Based on this property, Quinton shows that [29, 30] Equation (26) holds for all dependence vectors $J - I$ in $\mathcal{V}(S_1, S_2)$ if and only if the following two conditions hold:

$$\begin{aligned} &\text{for all vertex } V \text{ of } \mathcal{V}(S_1, S_2),\ TV > 0, \text{ and} \\ &\text{for all extremal ray } R \text{ of } \mathcal{V}(S_1, S_2),\ TR \geq 0. \end{aligned} \tag{30}$$

By Quinton's theorem, the input system C defined in Equation (29) can be simplified to:

$$\begin{aligned} \{TV > 0 \mid\ &\text{for all vertex } V \text{ of } \mathcal{V}(S_1, S_2), \text{ where } S_1 \text{ and } S_2 \\ &\text{are two statements in } \mathcal{S} \text{ such that } S_1 \Rightarrow S_2\} \\ \cup\ \{TR \geq 0 \mid\ &\text{for all extremal ray } R \text{ of } \mathcal{V}(S_1, S_2), \text{ where } S_1 \text{ and } S_2 \\ &\text{are two statements in } \mathcal{S} \text{ such that } S_1 \Rightarrow S_2\}. \end{aligned} \tag{31}$$

Consequently, the row vector T of length d, which defines an SSL-U schedule of a loop nest, can be obtained by linear programming, where d is the dimensionality of the index domain of the loop nest. The dimensionality of the linear programming system is d, and the number of constraints is the sum of the number of vertices and extremal rays.

5.2 An Algorithm for Statement Reordering

Having reviewed the algorithm for generating an SSL-U schedule with arbitrary statement reordering function r, we now discuss how to obtain an SSL-U schedule with statement reordering terms: $\pi(I,S) = (g_1(I), r(S))$, where g_1 is the temporal morphism defined in Equation (3), and r is the statement reordering function defined in Equation (4). The method can be modified easily to obtain the statement reordering functions for other single-sequential level schedules, i.e. SSL-SD, SSL-SV and SSL-NU.

The original input system for obtaining a schedule $\pi(I,S) = (g_1(I), r(S))$ should be:

$$\begin{aligned} C = \{(g_1(J), r(S_2)) \succ (g_1(I), r(S_1)) \mid & \text{ there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \\ & \text{and index tuples } I \text{ and } J \text{ in } D, \text{ such that } S_1@I \Rightarrow S_2@J\}. \end{aligned} \tag{32}$$

To solve for g_1 and r separately in two steps, the formulation is developed as follows. We start with the notion of minimal target difference vectors.

Minimal Target Difference Vector We define Γ to be the function space $[D \rightarrow E_1]$, where E_1 is a one-dimensional temporal index domain, and define $\mathcal{Z}$ to be the set of rationals. We define a second order function $\mu(f, S_1, S_2)$ to be the minimal target difference vector (in the sense of lexicographical ordering on elements of E_1) ranging over the image of the set of dependence vectors $\mathcal{V}(S_1, S_2)$ defined in Equation (28) under function $f \in \Gamma$:

$$\mu \ : \ \Gamma \times \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{Z} \tag{33}$$

$$\mu(f, S_1, S_2) \ = \ \min\{f(K) \mid K \in \mathcal{V}(S_1, S_2)\}. \tag{34}$$

It is easy to see that condition $\mu(g_1, S_1, S_2) > 0$ holds if and only if $g_1(J - I) > 0$ holds for all $(J - I) \in \mathcal{V}(S_1, S_2)$.

Input Systems for g_1 and r Due to the lexicographical ordering $\succ$, condition $(g_1(J), r(S_2)) \succ (g_1(I), r(S_1))$ in Equation (32) holds for all $(J - I) \in \mathcal{V}(S_1, S_2)$ if and only if either $\mu(g_1, S_1, S_2) > 0$ holds or both $\mu(g_1, S_1, S_2) = 0$ and $r(S_2) > r(S_1)$ hold. Consequently, as far as the statement reordering function r is concerned, the dependences $S_1 \Rightarrow S_2$ where $\mu(g_1, S_1, S_2) > 0$ do not provide any constraint on r. Only for those dependences $S_1 \Rightarrow S_2$ such that $\mu(g_1, S_1, S_2) = 0$, the conditions $r(S_1) < r(S_2)$ must hold. So the algorithm for generating SSL schedules can start out with a less stringent criterion $\mu(g_1, S_1, S_2) \geq 0$ for all pairs of statements S_1 and S_2 in $\mathcal{S}$ such that $S_1 \Rightarrow S_2$ to find g_1, and follow by the criterion $r(S_1) < r(S_1)$ for those dependence relation $S_1 \Rightarrow S_2$ where $\mu(g_1, S_1, S_2) = 0$.

From the above discussion, we know that the input system defined in Equation (32) can be separated into two parts, one for temporal morphism g_1 and the other for statement reordering function r:

$$\text{for } g_1 \ : \quad \begin{aligned} C_{g_1} = \{\mu(g_1, S_1, S_2) \geq 0 \mid & \text{ there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \\ & \text{such that } S_1 \Rightarrow S_2\}, \end{aligned} \tag{35}$$

$$\text{for } r: \quad C_r = \{r(S_1) < r(S_2) \mid \text{there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \text{ such that } S_1 \Rightarrow S_2 \text{ and } \mu(g_1, S_1, S_2) = 0\}. \tag{36}$$

The algorithm discussed in Section 5.1, and those to be discussed in Section 5.3, 5.4, and 5.5 can be used to obtain temporal morphism g_1 from the input system specified in Equation (35). We now discuss how to obtain statement reordering function r from the input system specified in Equation (36) given a temporal morphism g_1.

Partial Ordering and Topological Sort To obtain statement reordering function r given a temporal morphism g_1, we need the notion of *partial ordering* [16]. A partial order on a set S is a binary relation "$<$" that is *transitive* and *irreflexive*, i.e. there cannot be any cyclic relations $x < \ldots < x$ for all elements $x \in S$. For the purpose of statement reordering, we define S to be the set of statements $\mathcal{S}$ and we say $S_1 < S_2$, i.e. statement S_1 must be in front of statement S_2 in the transformed loop body, if $S_1 \Rightarrow S_2$ and $\mu(g_1, S_1, S_2) = 0$.

If $\mathcal{S}$ has a partial ordering, then *topological sort* can produce a linear ordering of the statements [16], which defines r. If $\mathcal{S}$ does not have a partial ordering, e.g. if there are cyclic relations $S_1 < S_2$ and $S_2 < S_1$, then there cannot be any r that will satisfy both $r(S_1) < r(S_2)$ and $r(S_1) > r(S_2)$. In this case, the given temporal morphism g_1 should be rejected.

5.3 Subdomain Scheduling Algorithm with Bounded Search Space

This section presents an algorithm for generating SSL-SD schedules. The hard part of finding a subdomain schedule for a given loop nest is to determine where the subdomain boundaries are. In this section, we present an algorithm which searches through a bounded space for possible hyperplanes that partition the domain, and comes up with affine schedules, one for each subdomain. Since the complexity of this method is too high, in Section 5.4 we describe a heuristic that first makes guesses at possible subdomain boundaries by unbounded *inside-out* enumerative search, and then obtains a subdomain schedule by linear programming with given subdomains.

5.3.1 Problem Formulation

Constraints Derived from Data Dependence For an SSL-SD schedule defined in Equation (8), the inequality

$$(T_j J, r_j(S_2)) - (T_i I, r_i(S_1)) \succ \hat{0} \tag{37}$$

must hold for all index tuples $I \in D_i$ and $J \in D_j$, $1 \leq i, j \leq m$, and statements S_1 and S_2 in $\mathcal{S}$ such that $S_1@I \Rightarrow S_2@J$, where T_i and T_j are row vectors of length d. We focus on the problem of obtaining T_i satisfying the condition

$$T_j J - T_i I = [-T_i, T_j] \begin{bmatrix} I \\ J \end{bmatrix} \geq 0, \tag{38}$$

which is the first step in obtaining an SSL-SD schedule. This step will be followed by a topological sort to find the statement reordering function as discussed in Section 5.2.

Since row vector T_i can be different from T_j, Equation (38) cannot be rewritten as $T(J-I) \geq 0$. Consequently, dependence vectors are not adequate for obtaining subdomain schedules. A new representation of a dependence relation is necessary.

Dependence Index pairs We now introduce a new notion of dependence relations. If dependence $S_1@I \Rightarrow S_2@J$ exists, then we call (I, J) a *dependence index pair* from statement S_1 to S_2.

Space of Dependence Index Pairs Similar to the set of dependence vectors $\mathcal{V}(S_1, S_2)$ defined in Equation (28), we formulate the set of dependence index pairs from statement S_1 to S_2, denoted by $\mathcal{P}(S_1, S_2)$, as:

$$\mathcal{P}(S_1, S_2) = \{(I, J) \mid I \in (D \downarrow P_1), J \in (D \downarrow P_2), \text{ and } X(I) = Y(J)\}. \tag{39}$$

It is easy to see that, under the restrictions discussed in Section 5.1, $\mathcal{P}(S_1, S_2)$ is a $(2d)$-dimensional polyhedron, where d is the dimensionality of the index domain of the loop nest.

Statement Reordering In Section 5.2, we discuss how to obtain a statement reordering function r given an SSL-U temporal morphism g_1. In fact, all formulations in Section 5.2 also hold for other classes of g_1, i.e. SSL-SD, SSL-SV and SSL-NU, except that we need to use the set of dependence index pairs $\mathcal{P}(S_1, S_2)$ to replace the set of dependence vectors $\mathcal{V}(S_1, S_2)$ in the formulation.

Let Γ, E_1, $\mathcal{Z}$ be as defined in Section 5.2. We define a second-order function $\mu(f, S_1, S_2)$ to be the minimal target difference vector ranging over the vectors $f(J) - f(I)$ where $(I, J) \in \mathcal{P}(S_1, S_2)$ and $f \in \Gamma$:

$$\mu(f, S_1, S_2) \;=\; \min\{f(J) - f(I) \mid (I, J) \in \mathcal{P}(S_1, S_2)\}. \tag{40}$$

Input System The input system for obtaining an SSL-SD temporal morphism g_1 consists of the following constraints:

$$\begin{aligned} C = \{[-T_i, T_j] \begin{bmatrix} I \\ J \end{bmatrix} \geq 0 \mid & \text{ there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \text{ such that} \\ & (I, J) \in Q(S_1, S_2, i, j)\}, \quad \text{where} \\ Q(S_1, S_2, i, j) = \{(I, J) \mid & (I, J) \in \mathcal{P}(S_1, S_2), I \in D_i, J \in D_j\}. \end{aligned} \tag{41}$$

Let B be a row vector of length d and c be a scalar. Let $\mathcal{Z}$ be the set of rationals. A *hyperplane* is a subspace of the d-dimensional vector space $\mathcal{Z}^d$ that can be expressed as $\{I \mid BI = c\}$. In order to obtain subdomain schedules systematically, we restrict that

the subdomains are separated by hyperplanes. Under this restriction, each subdomain D_i, $1 \leq i \leq m$, is a d-dimensional polyhedron, and each $Q(S_1, S_2, i, j)$ defined in Equation (41) is a $(2d)$-dimensional polyhedron.

Again, we need to represent each polyhedron $Q(S_1, S_2, i, j)$ by its vertices and extremal rays. However, since polyhedron $Q(S_1, S_2, i, j)$ defined in Equation (41) is specified by two unknown subdomains D_i and D_j, Algorithm ER of Section 5.1 is not directly applicable. So the key point is to do polyhedra decomposition under unknown constraints. Then the input system defined in Equation (41) can be simplified to another system with all dependence index pairs being replaced by vertices and extremal rays as discussed before. In the following, we first use a very simple example to show the basic idea of doing this. We then present the formal solution.

5.3.2 Basic Idea

We take the following loop whose index domain D is a one-dimensional interval domain $[l, u]$ as an example:

Loop Nest 14

$$
\begin{array}{l}
\mathsf{DO}\ (i = l, u)\ \{ \\
\quad S_1 :\ \mathsf{IF}(i \in [l_1, u_1])\ A(10) = \ldots \\
\quad S_2 :\ \mathsf{IF}(i \in [l_2, u_2])\ \ldots = A(10) \\
\quad \ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad \}
\end{array}
$$

where l, u, l_1, u_1, l_2 and u_2 are integer constants or variables under the conditions $[l_1, u_1] \subset [l, u]$ and $[l_2, u_2] \subset [l, u]$. For this example, $\mathcal{P}(S_1, S_2)$, defined in Equation (39), is a two-dimensional polyhedron:

$$\mathcal{P}(S_1, S_2) = \{(i, j) \mid i \in [l_1, u_1], j \in [l_2, u_2], 10 = 10\}, \tag{42}$$

which contains four vertices (l_1, l_2), (l_1, u_2), (u_1, l_2), and (u_1, u_2). Let D_1 and D_2

$$D_1 = D\downarrow(i \geq c) \quad \text{and} \tag{43}$$
$$D_1 = D\downarrow(i < c) \tag{44}$$

be two subdomains of D separated by the hyperplane $i = c$, where c is an unknown scalar. Under D_1 and D_2, polyhedron $\mathcal{P}(S_1, S_2)$ is partitioned into four disjoint parts $Q(S_1, S_2, 1, 1)$, $Q(S_1, S_2, 1, 2)$, $Q(S_1, S_2, 2, 1)$, and $Q(S_1, S_2, 2, 2)$ as defined in Equation (41), where

$$Q(S_1, S_2, 1, 1) = \{(i, j) \mid i \in [l_1, u_1], j \in [l_2, u_2], i \geq c, j \geq c\}, \tag{45}$$

and others have similar formulation. In the following we let $Q_{11} = Q(S_1, S_2, 1, 1)$.

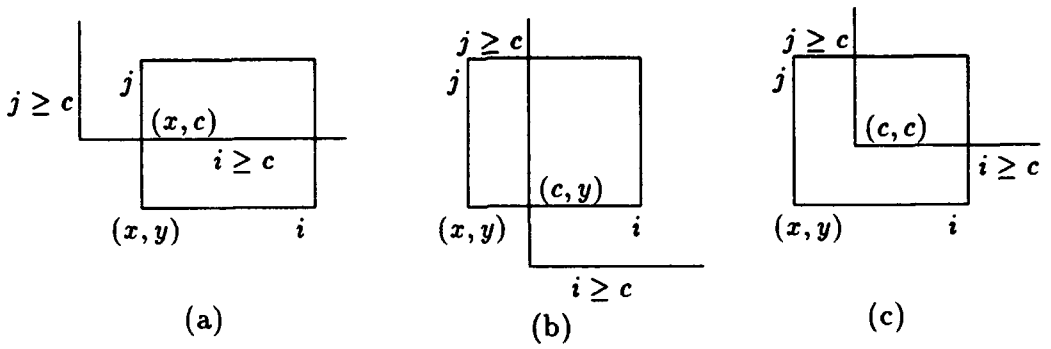

Figure 2: Bounded Search Space

Bounded Search One way to find the vertices of Q_{11} is to try all possible c values. For each c, the vertices of Q_{11} can be obtained by Algorithm ER of Section 5.1. However, there can be too many c to be searched, or even infinitely many if the loop bounds are unknown at compile time. In fact, for any c value, a vertex (x, y) of $\mathcal{P}(S_1, S_2)$ implies that

1. (x, y) is a vertex of Q_{11} if $x \geq c$ and $y \geq c$,
2. (x, c) is a vertex of Q_{11} if $x \geq c$ and $y < c \leq u_2$, as shown if Figure 2(a),
3. (c, y) is a vertex of Q_{11} if $x < c \leq u_1$ and $y \geq c$, as shown if Figure 2(b),
4. (c, c) is a vertex of Q_{11} if $x < c \leq u_1$ and $y < c \leq u_2$, as shown if Figure 2(c), or
5. Q_{11} has no vertex if $c > u_1$ or $c > u_2$.

Therefore, based on the relative values of x, y and c, there are four possible vertices of Q from a vertex (x, y) of $\mathcal{P}(S_1, S_2)$. Note that three of the four possible vertices of Q are parameterized with unknown scalar c. So the basic idea is that the search space of the partitioning hyperplanes can be bounded if parameterized vertices and extremal rays are used.

Nonlinear Programming With parameterized vertices and extremal rays, the input system defined in Equation (41) will become a nonlinear system. Hence bounded search and nonlinear programming are required to obtain the partitioning hyperplanes and the schedule for each disjoint subdomain.

We now present the formal solution. We first focus on the case when there is one partitioning hyperplane. The more general cases of multiple hyperplanes are discussed later.

5.3.3 One Partitioning Hyperplane

A partitioning hyperplane can be expressed as $BI = c$, where B is an unknown row vector of length d and c is an unknown scalar. In the case of one partitioning hyperplane, an SSL-SD schedule can be formulated as follows, again ignoring the statement reordering terms for the moment:

$$\pi(I,S) = \left\{ \begin{array}{l} (I \in D) \wedge (BI \geq c) \rightarrow T_1 I \\ (I \in D) \wedge (BI < c) \rightarrow T_2 I \end{array} \right\}. \tag{46}$$

According to subdomains $D_1 = D\downarrow(BI \geq c)$ and $D_2 = D\downarrow(BI < c)$, polyhedron $Q(S_1, S_2, 1, 1)$ defined in Equation (41) is:

$$Q_{11} = Q(S_1, S_2, 1, 1) = \mathcal{P}(S_1, S_2)\downarrow((BI \geq c) \wedge (BJ \geq c)). \tag{47}$$

Polyhedra Decomposition Under Unknown Constraints Again, we need to represent Q_{11} defined in Equation (47) by its vertices and extremal rays. As described in Section 5.1, this is equivalent to finding the extremal rays of polyhedron Q_h, which is the corresponding homogeneous polyhedron of Q_{11}:

$$Q_h = P_h\downarrow((B_1 \begin{bmatrix} K \\ h \end{bmatrix}) \geq 0) \wedge (B_2 \begin{bmatrix} K \\ h \end{bmatrix}) \geq 0)), \tag{48}$$

where P_h is the corresponding homogeneous polyhedron of $\mathcal{P}(S_1, S_2)$, and $B_1 \begin{bmatrix} K \\ h \end{bmatrix} \geq 0$ and $B_2 \begin{bmatrix} K \\ h \end{bmatrix} \geq 0$ are the homogeneous representations of $BI \geq c$ and $BJ \geq c$ respectively, i.e.

$$K = \begin{bmatrix} I \\ J \end{bmatrix}, \tag{49}$$

$$B_1 = [B, \hat{0}, -c], \text{ and} \tag{50}$$

$$B_2 = [\hat{0}, B, -c]. \tag{51}$$

Note that since row vector B and scalar c given in Equation (46) are unknown, row vectors B_1 and B_2 are also unknown. Recall that Algorithm ER of Section 5.1 iterates over the set of input constraints. Therefore, in obtaining the extremal rays of Q_h, we can first obtain the set of extremal rays of P_h, denoted by $\mathcal{R}$. Two more iterations are required to obtain the extremal rays of Q_h from $\mathcal{R}$, one for each unknown constraint $B_1 \begin{bmatrix} I \\ h \end{bmatrix} \geq 0$ and $B_2 \begin{bmatrix} J \\ h \end{bmatrix} \geq 0$.

Bounded Search Let $\mathcal{R}$ be the set of extremal rays of P_h, $\mathcal{R}_1$ be the set of extremal rays of $P_h \downarrow (B_1 \begin{bmatrix} I \\ h \end{bmatrix} \geq 0)$, and $\mathcal{R}_2$ be the set of extremal rays of Q_h defined in Equation (48). We have the following lemma:

Lemma 1 *The number of possible sets $\mathcal{R}_1$ and $\mathcal{R}_2$ derived from all possible row vectors B_1 and B_2 is bounded.*

Proof. From Algorithm ER, we know that the set $\mathcal{R}_1$ is obtained according to the projections of the rays in $\mathcal{R}$ on B_1. Although B_1 is unknown, the projection of a vector on any vector can only be either zero, positive or negative. Therefore, based on the signs of projections, if there are r rays in $\mathcal{R}$, then there are at most 3^r possible sets $\mathcal{R}_1$. And since a ray in $\mathcal{R}_1$ is generated from either a ray or a pair of rays in $\mathcal{R}$, each set $\mathcal{R}_1$ contains a finite number of rays. Let q be the maximal number of rays in all sets $\mathcal{R}_1$. Clearly, there are at most 3^{r+q} possible sets $\mathcal{R}_2$.

Nonlinear Formulation of Extremal Rays Since the values of the projections are unknown, some rays in $\mathcal{R}_1$ need to be specified by unknown scalars. For example, a ray in $\mathcal{R}_1$ generated from two rays R_1 and R_2 in $\mathcal{R}$ is a linear combination of R_1 and R_2:

$$\alpha_1 R_1 + \alpha_2 R_2, \tag{52}$$

where α_1 and α_2 are positive variables satisfying the nonlinear condition $B_1(\alpha_1 R_1 + \alpha_2 R_2) = 0$ as described in Algorithm ER.

Similarly, a ray of Q_h generated from two linear rays $(\alpha_1 R_1 + \alpha_2 R_2)$ and $(\alpha_3 R_3 + \alpha_4 R_4)$ in $\mathcal{R}_1$ is a nonlinear combination of R_i, $1 \leq i \leq 4$:

$$\alpha_5(\alpha_1 R_1 + \alpha_2 R_2) + \alpha_6(\alpha_3 R_3 + \alpha_4 R_4), \tag{53}$$

where α_5 and α_6 are positive variables satisfying the nonlinear condition $B_2(\alpha_5(\alpha_1 R_1 + \alpha_2 R_2) + \alpha_6(\alpha_3 R_3 + \alpha_4 R_4)) = 0$.

Nonlinear Constraints With unknown T_1, T_2, B_1, B_2 and α, we may have the following nonlinear inequalities that constitutes a system for obtaining an SSL-SD schedule defined in Equation (46):

$$[-T_1, T_2](\alpha_5(\alpha_1 R_1 + \alpha_2 R_2) + \alpha_6(\alpha_3 R_3 + \alpha_4 R_4)) \geq 0, \tag{54}$$

$$B_1(\alpha_1 R_1 + \alpha_2 R_2) = 0, \tag{55}$$

$$B_1(\alpha_3 R_3 + \alpha_4 R_4) = 0, \text{ and} \tag{56}$$

$$B_2(\alpha_5(\alpha_1 R_1 + \alpha_2 R_2) + \alpha_6(\alpha_3 R_3 + \alpha_4 R_4)) = 0. \tag{57}$$

Equation (54) is a constraint for T_1 and T_2, which specify the mapping of a subdomain schedule, and Equations (55), (56) and (57) are constraints for B and c, which specify the partitioning hyperplane of a subdomain schedule. Therefore, the order of the nonlinear constraints is three.

Nonlinear Systems To summarize, given one partitioning hyperplane, there is a bounded number of sets of parameterized extremal rays, where each extremal ray can be specified by a nonlinear expression of order two or less. Each set of parameterized extremal rays corresponds to a possible partition configuration. And we need to solve a nonlinear system of order three for each possible partition configuration to obtain T_1, T_2, B, and c, which define a subdomain schedule given in Equation (46). Clearly, many partition configurations would be invalid, and there can be one or more suitable partition configuration, or none.

From the above discussion, we have the following theorem:

Theorem 2 *An SSL-SD schedule with one partitioning hyperplane on the domain can be obtained by enumerative search through a bounded search space, at each instance of which nonlinear programming of order three is required to obtain the partitioning hyperplane and the schedule for each subdomain.*

5.3.4 Multiple Partitioning Hyperplanes

We now consider the cases with more than two subdomains separated by multiple partitioning hyperplanes. From the above case, it is easy to see that if there are p partitioning hyperplanes, then polyhedron Q_h defined in Equation (48) would be specified by $2p$ unknown constraints, and each extremal ray of Q_h would be of order $2p$ or less. Therefore, the order of the systems for obtaining an SSL-SD schedule would be $2p + 1$. We thus have the following corollary:

Corollary 3 *An SSL-SD schedule with p partitioning hyperplanes on the domain can be obtained by enumerative search through a bounded search space, at each instance of which nonlinear programming of order $2p + 1$ is required to compute the partitioning hyperplanes and the schedule for each subdomain.*

5.4 A Heuristic Algorithm for Generating Subdomain Schedules

Inside-Out Enumerative Search We now show that, if the subdomains of a subdomain schedule are separated by a set of given partitioning hyperplanes, then an SSL-SD schedule can be obtained by linear programming. We can use *inside-out* unbounded enumerative search [18, 19] as a heuristic to generate these partitioning hyperplanes. Let a partitioning hyperplane be specified by $BI = c$, where B is a row vector, and c is a scalar. The inside-out enumerative search starts by setting the bounds of the absolute values of the elements of row vector B and scalar c to be 1, then 2, 3, etc., until a schedule is found. Hopefully, if there exists a subdomain schedule, the values of the elements of B and c would be small and found quickly. And this appears to be the case for many example programs including the dynamic programming example to be discussed in Section 8.

Linear Programming Formulation If all partitioning hyperplanes are given, then each polyhedron $Q(S_i, S_2, i, j)$ defined in Equation (41) can be decomposed into vertices and

extremal rays by Algorithm ER of Section 5.1. Therefore, the input system C defined in Equation (41) can be simplified to the following form:

$$\begin{aligned} &\{[-T_i, T_j]V \geq 0 \mid \text{for all vertex } V \text{ of } Q(S_1, S_2, i, j), \text{ where } S_1 \text{ and } S_2 \\ &\qquad\qquad \text{are two statements in } \mathcal{S} \text{ such that } S_1 \Rightarrow S_2\} \\ &\cup \{[-T_i, T_j]R \geq 0 \mid \text{for all extremal ray } R \text{ of } Q(S_1, S_2, i, j), \text{ where } S_1 \text{ and } S_2 \\ &\qquad\qquad \text{are two statements in } \mathcal{S} \text{ such that } S_1 \Rightarrow S_2\}. \end{aligned} \tag{58}$$

Consequently, m row vectors T_i, $1 \leq i \leq m$, of length d, which define an SSL-SD schedule, can be obtained by linear programming. The dimensionality of the linear programming system is $m * d$, where m is the number of subdomains and d is the dimensionality of the index domain of the loop nest. We thus have the following theorem:

Theorem 4 *Given the partitioning hyperplanes of the index domain D of a loop nest, an SSL-SD schedule can be obtained by linear programming. The dimensionality of the linear programming system is $m * d$, where m is the number of subdomains and d is the dimensionality of the index domain of the loop nest.*

5.5 An Algorithm for Generating Statement-Variant Schedules

Single Statement in Each Partition From the subdomain schedule, we know that it is hard to find the partition of a domain and the schedule for each subdomain at the same time. If the partition of the domain is given, then the problem becomes much simpler. This is also true for obtaining statement-variant schedules. Let s be the number of statements in a loop body. If s is not too large, then we can consider the extreme case of a statement-variant schedule that each partition contains only one statement. In this case, the input system C containing all constraints is:

$$\begin{aligned} C &= \{[-T_i, T_j]\begin{bmatrix} I \\ J \end{bmatrix} \geq 0 \mid \text{there exist statements } S_i \text{ and } S_j \text{ in } \mathcal{S}, \text{ and index tuples} \\ &\qquad\qquad I \text{ and } J \text{ in } D \text{ such that } S_i@I \Rightarrow S_j@J\}. \\ &= \{[-T_i, T_j]\begin{bmatrix} I \\ J \end{bmatrix} \geq 0 \mid \text{there exist statements } S_i \text{ and } S_j \text{ in } \mathcal{S}, \text{ such that} \\ &\qquad\qquad (I, J) \in \mathcal{P}(S_i, S_j)\}, \end{aligned} \tag{59}$$

where $\mathcal{P}(S_i, S_j)$ is defined in Equation (39).

Linear Programming Formulation Since the vertices and extremal rays of $\mathcal{P}(S_i, S_j)$ can be obtained by by Algorithm ER of Section 5.1, the input system C defined in Equation

(59) can be simplified to:

$$\begin{aligned} &\{[-T_i, T_j]V \geq 0 \mid \text{for all vertex } V \text{ of } \mathcal{P}(S_i, S_j), \text{ where } S_i \text{ and } S_j \\ &\qquad\qquad \text{are two statements in } \mathcal{S} \text{ such that } S_i \Rightarrow S_j\} \\ &\cup \{[-T_i, T_j]R \geq 0 \mid \text{for all extremal ray } R \text{ of } \mathcal{P}(S_i, S_j), \text{ where } S_i \text{ and } S_j \\ &\qquad\qquad \text{are two statements in } \mathcal{S} \text{ such that } S_i \Rightarrow S_j\}. \end{aligned} \tag{60}$$

Consequently, s row vectors T_i, $1 \leq i \leq s$, of length d, which define an SSL-SV schedule, can be obtained by linear programming. We thus have the following theorem:

Theorem 5 *An SSL-SV schedule of a loop nest, where each partition of statements contains only one statement, can be obtained by linear programming. The dimensionality of the linear programming system is $s*d$, where s is the number of statements and d is the dimensionality of the index domain of the loop nest.*

Algorithm for Generating Nonuniform Schedules Since a nonuniform schedule is a combination of a subdomain schedule and a statement-variant schedule, it can be obtained by the algorithms described in Sections 5.3, 5.4 and 5.5.

6 An Iterative Algorithm for Generating Multiple-Sequential Level Schedules

We now present the algorithm for generating multiple-sequential level schedules. To obtain an n-sequential level schedule, our approach is to first find n SSL temporal morphisms $g_1^1, \ldots, g_1^n$, one at a time, followed by generating the statement reordering function r. This technique applies to all schedules: uniform, subdomain-variant, statement-variant or nonuniform.

To describe an iterative algorithm, we need to define a sequence of minimal target difference vectors similar to the ones defined in Equations (34) and (40).

A Sequence of Target Difference Vectors We first define Γ_i, $1 \leq i \leq n$, to be the function space $[D \rightarrow E_1^1 \times \ldots \times E_1^i]$, where each E_1^j, $1 \leq j \leq n$, is a one-dimensional temporal index domain, and define $\mathcal{Z}$ to be the set of rationals.

For uniform schedules, we define a family of second-order functions $\mu_i(f, S_1, S_2)$, $1 \leq i \leq d$, to be a sequence of minimal target difference vectors (in the sense of lexicographical ordering on elements of $E_1 \times \ldots \times E_1^i$) ranging over the images of the set of dependence vectors $\mathcal{V}(S_1, S_2)$ defined in Equation (28) under function $f \in \Gamma_i$:

$$\begin{aligned} \mu_i &: \Gamma_i \times \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{Z}, \text{ where} \\ \mu_i(f, S_1, S_2) &= \min\{f(K) \mid K \in \mathcal{V}(S_1, S_2)\}. \end{aligned} \tag{61}$$

For subdomain-variant, statement-variant and nonuniform schedules, we define a family of second-order functions $\mu_i(f, S_1, S_2)$, $1 \leq i \leq d$, to be a sequence of minimal target difference vectors ranging over the vectors $f(J) - f(I)$ where $(I, J) \in \mathcal{P}(S_1, S_2)$ and $f \in \Gamma_i$:

$$\mu_i(f, S_1, S_2) = \min\{f(J) - f(I) \mid (I, J) \in \mathcal{P}(S_1, S_2)\}. \quad (62)$$

Iterative Algorithm Similar to the algorithm for obtaining SSL schedules with statement reordering functions as discussed in Section 5.2, the algorithm for generating multiple-sequential level schedules starts with criterion $\mu(g_1^1, S_1, S_2) \geq 0$ for all pairs of statements S_1 and S_2 such that $S_1 \Rightarrow S_2$ to generate an SSL temporal morphism g_1^1. We then use the constraint $\mu(g_1^2, S_1, S_2) \geq 0$ for those dependences $S_1 \Rightarrow S_2$ such that $\mu(g_1^1, S_1, S_2) = 0$ to find g_1^2. The same process is iterated until the n-th iteration, $1 \leq n \leq d$, when the remaining dependences $S_1 \Rightarrow S_2$ where $\mu_n((g_1^1, \ldots, g_1^n), S_1, S_2) = \hat{0}$ are not cyclic. In this case, the set of statements $\mathcal{S}$ has the partial ordering defined in Section 5.2. We then use topological sort to find the linear ordering of the statements.

We summarize the above discussion as the following iterative algorithm for generating multiple-sequential level schedules: The inputs of Algorithm MSL consist of the set of all dependences in a loop nest, denoted by $\mathcal{L}$, and the choice of one of the algorithms presented in Section 5.1, 5.3, 5.4 or 5.5, denoted by $\mathcal{A}$.

Algorithm MSL ($\mathcal{L}$: a set of dependences, $\mathcal{A}$: an algorithm)

1. $i \leftarrow 0$ (i will be ranging over the loop levels);
2. While $i \leq d$ and $\mathcal{L}$ contains cyclic dependences, do
   (a) $i \leftarrow i + 1$;
   (b) Find an SSL temporal morphism g_1^i by using algorithm $\mathcal{A}$ such that $\mu(g_1^i, S_1, S_2) \geq 0$ holds for all $S_1 \Rightarrow S_2 \in \mathcal{L}$;
   (c) If such g_1^i does not exist, then the algorithm terminates without returning a schedule;
   (d) Else, $\mathcal{L} \leftarrow \{(S_1 \Rightarrow S_2) \mid (S_1 \Rightarrow S_2) \in \mathcal{L}$ such that $\mu(g_1^i, S_1, S_2) = 0\}$;
3. (Now $\mathcal{L}$ contains no cyclic dependences.) Find a statement reordering function r by topological sort, such that $r(S_1) < r(S_2)$ for all $(S_1 \Rightarrow S_2) \in \mathcal{L}$.

Note that the sequential loops in the transformed loop nest generated by Algorithm MSL are perfectly nested.

7 A Recursive Algorithm for Generating Mixed Schedules

We now present the algorithm for generating mixed schedules. We first discuss the basic idea of this algorithm.

Basic Idea Algorithm MSL presented in the previous section for generating multiple-sequential level schedules treats all the statements in the same way throughout the iterations even though more and more of the dependences become uninformative in obtaining the sequence $g_1^1, \ldots, g_1^n$. Clearly, when some instances of dependence relations are not considered, a new set of equivalence classes under relation "$\sim$" over statements emerges. As discussed in Section 2, these new equivalence classes can be scheduled separately for the subsequent iterations. To do this, a tree of temporal morphisms, instead of just a sequence $g_1^1, \ldots, g_1^n$, will be generated.

Labeling a Tree We use a prefix notation to label each node of the trees of temporal morphisms, equivalence classes and sets of dependence relations to be defined later. Let the root of the tree be labeled 1. For a node which is the i-th child of its parent node, we say the *order* of this node is i. A node is labeled $l \circ i$ if l is the label of its parent node and it has order i. A temporal morphism g_1 with label l is denoted by $g_1(l)$, an equivalence class B with label l is denoted by $B(l)$, and a set of dependences $\mathcal{L}$ with label l is denoted by $\mathcal{L}(l)$.

Recursive Algorithm The initial inputs of the recursive algorithm for generating mixed schedules consist of the equivalence class $B(1)$, which contains all statements in $\mathcal{S}$, and the set $\mathcal{L}(1)$, which contains all dependences in the source loop nest. With inputs $B(l)$ and $\mathcal{L}(l)$, the algorithm obtains an SSL temporal morphism $g_1(l)$ such that $\mu(g_1(l), S_1, S_2) \geq 0$ holds for all $(S_1 \Rightarrow S_2) \in \mathcal{L}(l)$. Under $g_1(l)$, the set of dependences

$$\mathcal{U}(l) = \{(S_1 \Rightarrow S_2) \mid (S_1 \Rightarrow S_2) \in \mathcal{L}(l), \mu(g_1(l), S_1, S_2) > 0\} \tag{63}$$

becomes uninformative and should be removed from $\mathcal{L}(l)$. A new set of equivalence classes, denoted by $\mathsf{new}(l)$, emerges:

$$\mathsf{new}(l) = \{B(l \circ i) \mid B(l \circ i) \text{ is the } i\text{-th equivalence class from } \mathcal{L}(l) - \mathcal{U}(l)\}. \tag{64}$$

For each new equivalence class $B(l \circ i)$, $1 \leq i \leq |\mathsf{new}(l)|$, the associated set of dependences $\mathcal{L}(l \circ i)$ is

$$\mathcal{L}(l \circ i) = \{(S_1 \Rightarrow S_2) \mid S_1 \in B(l \circ i), S_2 \in B(l \circ i), (S_1 \Rightarrow S_2) \in (\mathcal{L}(l) - \mathcal{U}(l))\}. \tag{65}$$

If $B(l \circ i)$, $1 \leq i \leq |\mathsf{new}(l)|$, is a dependent block, then the same algorithm is applied recursively to the new inputs $B(l \circ i)$ and $\mathcal{L}(l \circ i)$. The recursive algorithm stops when all dependent blocks have been broken up and all remaining blocks are independent.

We summarize the above discussion as the following recursive algorithm for generating mixed schedules: The initial inputs of Algorithm MIX consist of the equivalence class $B(1)$, the set of dependences $\mathcal{L}(1)$, and the choice of one of the algorithms presented in Section 5.1, 5.3, 5.4 or 5.5, denoted by $\mathcal{A}$.

Algorithm MIX ($B(l)$: an equivalence class, $\mathcal{L}(l)$: a set of dependences, $\mathcal{A}$: an algorithm)

1. Find an SSL temporal morphism $g_1(l)$ by using algorithm $\mathcal{A}$, such that $\mu(g_1(l), S_1, S_2) \geq 0$ holds for all $(S_1 \Rightarrow S_2) \in \mathcal{L}(l)$;

2. If such $g_1(l)$ does not exist, then the algorithm terminates without returning a schedule.

3. obtain the set new(l) (remove uninformative dependences and generate new equivalence classes);

4. Generate new sets of dependences $\mathcal{L}(l \circ i)$, $1 \leq i \leq |\text{new}(l)|$;

5. For $i \in [1, |\text{new}(l)|]$, if $B(l \circ i)$ is a dependent block, then MIX$(B(l \circ i), \mathcal{L}(l \circ i), \mathcal{A})$.

Statement Reordering We now discuss how to obtain the statement reordering function r for mixed schedules. Since the algorithm for generating mixed schedules is recursive, statement ordering is also obtained recursively. Suppose we want to find the ordering among the statements in an equivalence class $B(l)$. Let $B(l \circ i)$, $1 \leq i \leq n$, be the equivalence classes in new(l) generated from $B(l)$. We can first determine the ordering among these n equivalence classes, which induces the ordering among statements in different $B(l \circ i)$, but not the ordering among statements within the same $B(l \circ i)$. Since each $B(l \circ i)$, $1 \leq i \leq n$, will be scheduled separately, the ordering among statements in different $B(l \circ i)$ will never be changed subsequently. The same process is then applied recursively to each $B(l \circ i)$, $1 \leq i \leq n$, to determine the ordering among statements within $B(l \circ i)$.

The ordering among these n equivalence classes is obtained as follows. Similar to Section 5.2, we define a partial ordering "$<$" over the set of equivalence classes new(l). We say $B(l \circ 1) < B(l \circ 2)$, i.e. equivalence class $B(l \circ 1)$ must be in front of equivalence class $B(l \circ 2)$ in the transformed loop body, if there exist statements $S_1 \in B(l \circ 1)$ and $S_2 \in B(l \circ 2)$ such that $(S_1 \Rightarrow S_2) \in \mathcal{L}(l)$ and $\mu(g_1(l), S_1, S_2) = 0$.

Due to the way these n equivalence classes are generated from $B(l)$, new(l) always has this partial ordering. Therefore, topological sort can produce the linear ordering of these n equivalence classes.

Imperfect Loop Nests For a given statement S, it may belong to a nested level of dependent blocks $B(1)$, ..., $B(l)$, where $B(1) \supset \ldots \supset B(l)$. Clearly, the schedule of statement S is $|l|$-sequential level, where $|l|$ is the length of the label l:

$$\pi(I, S) = (g_1(1)(I), \ldots, g_1(l)(I), r(S)). \tag{66}$$

Since different statements can belong to different sets of dependent blocks, the sequential level of $\pi(I, S)$ can be different for different statements. And the sequential loops in the transformed parallel loop nest can be of any form, i.e. perfectly and imperfectly nested loops.

When different equivalence classes are scheduled differently, each SSL temporal morphism will be a statement-variant schedule with the partition of statements being these new equivalence classes. Recall that in Section 5.5, a statement-variant schedule is obtained by allowing each statement to be scheduled differently, but not independently. The advantage to schedule independently is that the likelihood of obtaining a suitable schedule with more parallelism increases if each dependent block is considered separately.

Example We use the following example to explain Algorithm MIX further. In this example, each SSL temporal morphism used in Algorithm MIX is a uniform one.

Loop Nest 15

$$
\begin{array}{l}
\mathsf{DO}\ (i = 2, n)\,\{ \\
\quad \mathsf{DO}\ (j = 2, n)\,\{ \\
\quad\quad \mathsf{DO}\ (k = 2, n)\,\{ \\
\quad\quad\quad S_1 :\ A(i,j,k) = A(i-1,k,j) + B(i-1,j,k) \\
\quad\quad\quad S_2 :\ B(i,j,k) = B(i,j-1,j) + C(i-1,j,k) \\
\quad\quad\quad S_3 :\ C(i,j,k) = C(i,j,k-1) + A(i,j,k) \qquad \}\,\}\,\}
\end{array}
$$

The initial $B(1)$ is the set $\{S_1, S_2, S_3\}$ and $\mathcal{L}(1)$ is the set $\{(S_x \Rightarrow S_y) \mid (x,y) \in \{(1,1),(2,2),(3,3),(2,1),(3,2),(1,3)\}$. It is easy to check that if $g_1(1)((i,j,k),S) = i$ for all $S \in B(1)$, then $\mu(g_1(1), S_x, S_y) = 1$ for (x,y) in the set $U = \{(1,1),(2,1),(3,2)\}$. And $\mu(g_1(1), S_x, S_y) = 0$ for (x,y) in $(x,y) \in \{(2,2),(3,3),(1,3)\}$. Consequently, dependences $S_x \Rightarrow S_y$, $(x,y) \in U$, can be removed and $B(1)$ is broken up into three equivalence classes $B(1 \circ i)$, $1 \leq i \leq 3$; each contains a single statement S_i.

Since $B(1 \circ 1)$ is an independent block, the schedule for S_1 will be single-sequential level. Furthermore, S_2 and S_3 are in different equivalence classes and they can be scheduled independently. Let $g_1(1 \circ 2)((i,j,k),S) = j$ and $g_1(1 \circ 3)((i,j,k),S) = k$. It is easy to check that $\mu(g_1(1 \circ 2), S_2, S_2) = 1$ and $\mu(g_1(1 \circ 3), S_3, S_3) = 1$.

To summarize, the mixed schedule is single-sequential level for S_1 (two-dimensional parallelism), and is two-sequential level for S_2 and S_3 (one-dimensional parallelism) as shown below:

$$
\pi((i,j,k),S) = \left\{ \begin{array}{l} S = S_1 \rightarrow (i,0) \\ S = S_2 \rightarrow (i,j,1) \\ S = S_3 \rightarrow (i,k,2) \end{array} \right\}. \tag{67}
$$

Because $\mu(g_1(1), S_1, S_3) = 0$ and statements S_1 and S_3 are in different equivalence classes, S_1 must be in front of S_3 in the transformed loop body. The resulting parallelized program with imperfectly nested sequential loops is:

Loop Nest 16

DO $(i = 2, n)\{$
  DOALL $((j = 2, n), (k = 2, n))\{$
    $S_1 : A(i,j,k) = A(i-1,k,j) + B(i-1,j,k)\}$
  DO $(j = 2, n)\{$
    DOALL $(k = 2, n)\{$
      $S_2 : B(i,j,k) = B(i,j-1,j) + C(i-1,j,k)\}\}$
  DO $(k = 2, n)\{$
    DOALL $(j = 2, n)\{$
      $S_3 : C(i,j,k) = C(i,j,k-1) + A(i-1,j,k)\}\}\}$

8 An Application: Dynamic Programming

To illustrate the usefulness of the new scheduling algorithms, we take dynamic programming as an example, which has sequential complexity $O(n^3)$ for a problem of size n. The source code is given in Program 17.

Loop Nest 17

DO $(i = 1, n-1)\{$
  $C(i, i+1) = \text{initial} - \text{values}\}$
DO $(i = 1, n-2)\{$
  DO $(j = i+2, n)\{$
    $C(i,j) = \min_{i<k<j}(h(C(i,k), C(k,j)))\}\}$

This source program is first transformed to the following form in a systematic manner by applying *fan-in* and *fan-out* reductions [10] to reduce potential concurrent accesses of variables. The result is Program 18:

Loop Nest 18

DO $(i = 1, n-1)\{$
  DO $(k = 1, n)\{$
    $C(i, i+1, k) = \text{initial} - \text{values}\ \}\ \}$
DO $(i = n-1, 1, -1)\{$
  DO $(j = i+1, n)\{$
    $m = (i + j + 1)/2$
    DO $(k = i, j)\{$
      S_{a1} : IF$(k < j)\ A(i,j,k) = A(i,j-1,k)$
      S_{b1} : IF$(i+1 = k)\ B(i,j,k) = C(i+1,j,j)$
      S_{b2} : IF$(i+1 < k)\ B(i,j,k) = B(i+1,j,k)$
      S_{c1} : IF$(m = k)\ C(i,j,k) = h_1(A(i,j,k), B(i,j,k),$
        $A(i,j,i+j-k), B(i,j,i+j-k))$
      S_{c2} : IF$(m < k < j)\ C(i,j,k) = h_2(C(i,j,k-1), A(i,j,k),$
        $B(i,j,k), A(i,j,i+j-k), B(i,j,i+j-k))$
      S_{c3} : IF$(k = j)\ C(i,j,k) = C(i,j,k-1)$
      S_{a2} : IF$(k = j)\ A(i,j,k) = C(i,j,k)$ $\}\ \}\ \}$

Schedules We wrote three *lisp programs on the Connection Machine CM/2, each with the control structure generated by a two-sequential level uniform schedule, a mixed statement-variant schedule and a single-sequential level subdomain schedule respectively. We also have a sequential Common-Lisp program on the Symbolics to compute the same problem. The three schedules are given below. For simplicity, we do not give the constant terms $r(S)$ of function π.

two-sequential level uniform schedule:

$$\pi(S,(i,j,k)) = (j-i, k-i)$$

mixed statement-variant schedule:

$$\pi(S,(i,j,k)) = \left\{ \begin{array}{l} S = S_{c2} \rightarrow (j-i, k-i) \\ \textbf{else} \rightarrow j-i \end{array} \right\}$$

single-sequential level subdomain schedule:

$$\pi(S,(i,j,k)) = \left\{ \begin{array}{l} i+j-2k < 0 \rightarrow -2i+j+k \\ i+j-2k \geq 0 \rightarrow -i+2j-k \end{array} \right\}.$$

Experimental Result The experiment is conducted as follows: we run the sequential code on the Symbolics and parallel codes on an 8K-processor Connection Machine with

n	3-sequential level sequential	2-sequential level uniform	mixed statement-variant	1-sequential level subdomain
32	6.8	10.72	2.47	0.87
64	55.0	42.88	9.73	1.73
128	440.0	171.50	39.16	3.48
256	3520.0	686.45	235.70	6.96
512	28160.0	2745.80	1159.24	31.70

Figure 3: Running time in seconds.

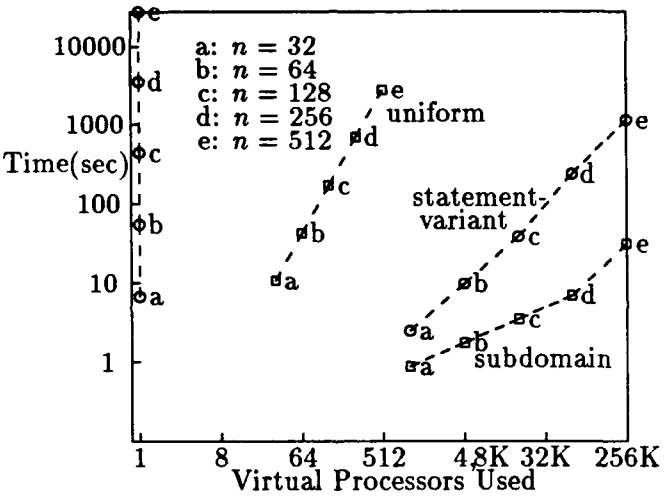

Figure 4: Running time vs. problem size.

Symbolics as its host. The results described in Figure 3 and Figure 4 show that the version using a single-sequential level subdomain schedule is three orders of magnitude faster than the sequential code, and is two orders of magnitude faster than the versions using a two-sequential level uniform schedule and mixed statement-variant schedule. And the program using a mixed statement-variant schedule is about three to four times faster than the program using a two-sequential level uniform schedule.

References

[1] J.R. Allen. *Dependence Analysis for Subscript Variables and Its Application to Program Transformation.* PhD thesis, Rice University, April 1983.

[2] J.R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, pages 233–246. ACM, 1984.

[3] J.R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[4] U. Banerjee. Data dependence in ordinary programs. Master's thesis, University of Illinois at Urbana-Champaign, November 1976.

[5] U. Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, October 1979.

[6] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[7] U. Banerjee. A theory of loop permutation. Technical report, Intel Corporation, 1989.

[8] U. Banerjee. Unimodular transformations of double loops. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing.* UC. Irvine, 1990.

[9] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 162–175. ACM, 1986.

[10] M.C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, December 1986.

[11] M.C. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 1(2):171–207, July 1988.

[12] J.M. Delosme and I.C.F. Ipsen. Systolic array synthesis: Computability and time cones. Technical Report RR-474, Yale University, 1986.

[13] F. Fernandez and P. Quinton. Extension of Chernikova's algorithm for solving general mixed linear programming problems. Technical Report 437, INRIA-Rennes, October 1988.

[14] C. Guerra and R. Melham. Synthesizing non-uniform systolic designs. In *Proceedings of the 1986 Int'l. Conf. on Parallel Processing*, pages 765–772. IEEE and ACM, 1986.

[15] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[16] D.E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, 1968.

[17] S. Kung. On supercomputing with systolic/wavefront array processors. *Proceedings of the IEEE*, 72(7):867–884, July 1984.

[18] P.Z. Lee and Z.M. Kedem. Mapping nested loop algorithms into multidimensional systolic arrays. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):64–76, January 1990.

[19] G.J. Li and Wah B.W. The design of optimal systolic arrays. *IEEE Transactions on Computer*, C-34(1):66–77, January 1985.

[20] Z. Li. Intraprocedural and interprocedural data dependence analysis for parallel computing. Technical Report 910, University of Illinois at Urbana-Champaign, August 1989.

[21] Z. Li, P.C. Yew, and C.Q. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, January 1990.

[22] W.M. Lin and V.K.P. Kumar. A note on the linear transformation method for systolic array design. *IEEE Transactions on Computer*, C-39(3):393–399, March 1990.

[23] L.C. Lu and M.C. Chen. Subdomain dependence test for massively parallel computing. In *Proc. Supercomputing '90*, 1990. to appear in.

[24] W.L. Miranker. Spacetime representations of computational structures. In *Computing*, volume 32, pages 93–114, 1984.

[25] D.I. Moldovan. On the analysis and synthesis of vlsi algorithms. *IEEE Trans. on Computers*, C-31(11):1121–1126, Nov. 1982.

[26] D.I. Moldovan. On the design of algorithms for vlsi systolic arrays. In *Proceedings of the IEEE*, volume 71, 1983.

[27] D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. on Computers*, C-35(1), Jan. 1986.

[28] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of 11th Annual Symposium on Computer Architecture*, pages 208–214, 1984.

[29] P. Quinton. Mapping recurrences on parallel architectures. In *Proceedings of the Third International Conference on Supercomputing*, pages 1–8. ICS, 1988.

[30] P. Quinton and V.V. Dongen. The mapping of linear recurrence equations on regular arrays. Technical Report 485, INRIA-Rennes, July 1989.

[31] S.K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Stanford University, October 1985.

[32] A. Schrijver. *Theory of Linear and Integer Programming. Wiley-Interscience series in Discrete Mathematics.* John Wiley and Sons, 1986.

[33] M.E. Wolf and M.S. Lam. Maximizing parallelism via loop transformations. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing.* UC. Irvine, 1990.

[34] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, October 1982.

[35] M. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 536–543, August 1986.

[36] M. Wolfe. More iteration space tiling. In *Proc. Supercomputing '89*, November 1989.

[37] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.

Contents